Extraction of ownership object graphs from object-oriented code: An experience report

Article · January 2012

DOI: 10.1145/2304696.2304719

3 authors, including:

Nariman Ammar
Wayne State University
14 PUBLICATIONS **34** CITATIONS
SEE PROFILE

Zeyad Hailat
Wayne State University
4 PUBLICATIONS **7** CITATIONS
SEE PROFILE

Extraction of Ownership Object Graphs from Object-Oriented Code: an Experience Report

Marwan Abi-Antoun Nariman Ammar Zeyad Hailat
Department of Computer Science
Wayne State University
{mabiantoun, nammar, zmhailat}@wayne.edu

ABSTRACT

Despite receiving much research attention, the extraction of runtime architecture remains hard. One approach, SCHOLIA, relies on adding typecheckable annotations to the code, and uses static analysis to extract a global, hierarchical Ownership Object Graph (OOG). The OOG provides architectural abstraction by ownership hierarchy and by types, and can be abstracted into a run-time architecture represented in an architectural description language, for documentation or conformance analysis.

We report on our experience in analyzing a medium-sized object-oriented system undergoing maintenance to: (1) extract an OOG; and (2) refine the OOG based on the maintainers' feedback.

We evaluate the effectiveness of abstraction by ownership hierarchy and by types to extract an OOG that the system maintainers understand. We measure the extraction effort to be about 1 hour/KLOC. An evaluation with the lead maintainer confirms that he understands abstraction by ownership hierarchy and by types. Finally, we illustrate how to incrementally refine an extracted OOG (without starting all over) to better match the maintainer's mental model.

Categories and Subject Descriptors

D.2.11 [**Software Engineering**]: Software Architectures

General Terms

Experimentation, Documentation

Keywords

runtime architecture, architecture recovery, ownership types

1. INTRODUCTION AND MOTIVATION

During software evolution, high-level architectural diagrams showing the system's organization are useful to maintainers. Unfortunately, architectural diagrams are missing or out-of-date for many legacy object-oriented systems undergoing maintenance. Because the most reliable and accurate description of an evolving system is its source code [23], one can use architectural extraction techniques to reverse-engineer from the code high-level diagrams of a system for redocumentation, conformance and analysis [12].

Many architectural diagrams are needed to describe a software system. The *code architecture* organizes code entities in terms of classes and packages [10], and is useful for studying properties such as maintainability. Another useful diagram, the *runtime architecture* [10], models runtime entities and their potential interactions. A runtime architecture is equally important as the code architecture, because it impacts quality attributes such as security and performance.

Despite receiving much research attention, extracting runtime architectures remains a hard problem. Many have preferred dynamic analysis (or mixed with static analysis) to extract the as-built runtime architecture [20, 21]. The only approach, currently available, to extract a hierarchical runtime architecture entirely statically is SCHOLIA by Abi-Antoun and Aldrich [3]. SCHOLIA is a two-pronged approach which requires first adding ownership annotations in the code to specify some architectural intent [6]. It then uses a static analysis to extract a hierarchical object graph, the Ownership Object Graph (OOG). The OOG provides architectural abstraction by ownership hierarchy and by types, where architecturally significant objects are near the top of the hierarchy and data structures are further down. Moreover, the OOG is *sound* in two respects. First, each runtime object has exactly one representative in the OOG. Second, the OOG has edges that correspond to all possible runtime points-to relations between those objects.

The OOG corresponds to a runtime architecture which abstracts objects into components, abstracts relations between objects into connections, and optionally decomposes a component into a nested sub-architecture. Instead of objects being directly inside other objects, the OOG has an extra level of hierarchy and groups related objects inside a *domain*. A domain is similar to an architectural runtime *tier*, i.e., a *conceptual partitioning of functionality* [10].

In this paper, we report on our experience in analyzing a medium-sized, object-oriented system undergoing maintenance to: (1) extract an OOG; and (2) refine the OOG based on the maintainers' input.

Research Questions. We refer to the person who added annotations to the code, extracted OOGs and refined them as the *extractor* (this paper's third author). One of the maintainers of the subject system evaluated the OOG (and is

QoSA'12, June 25–28, 2012, Bertinoro, Italy.

not a co-author). We posed the following research questions:

RQ1– Can the extractors effectively use abstraction by ownership hierarchy and by types to extract an OOG that conveys architectural abstraction? And how much effort does it take?

RQ2– Can the maintainers understand the OOG, i.e., abstraction by ownership hierarchy and by types?

RQ3– Can the extractors incrementally refine the OOG to make it convey the maintainers' design intent?

Contributions. This paper contributes the following:

- A careful tracking of the effort to apply abstraction by ownership hierarchy and by types to make an extracted OOG convey architectural abstraction, and a confirmation that the effort is about 1 hour/KLOC;
- An evaluation of the extracted OOG by the lead maintainer of the system, and a confirmation that he understood abstraction by ownership hierarchy and by types.
- A careful tracking of the effort needed to incrementally refine an extracted OOG (without starting all over) to better match the maintainer's mental model of the system, and a confirmation that the refinement effort is lower than the initial extraction effort.

Outline. This paper is organized as follows. Section 2 provides some background on the annotations and the OOG. Section 3 describes the subject system. Section 4 describes how we extracted and refined the OOG on our own before showing it to the maintainer. Section 5 discusses the feedback from the maintainer and the additional refinement of the OOG. Section 6 discusses some lessons learned. Finally, we discuss related work (Section 7) and conclude.

2. BACKGROUND

2.1 Ownership domain annotations

The annotations implement a type system, Ownership Domains [6], and are checked using a typechecking tool. An *ownership domain* is a conceptual group of objects with an explicit name that conveys design intent [6]. The concrete annotation language uses support for annotations available as of Java 1.5 [1, Appendix A has the full language]. Here, we illustrate the annotations on a small example (Fig. 1).

Domain declaration. The extractor must declare a domain before use, using the `@Domains` annotation (line 1). Domains are declared on a class but are treated like fields, in that fresh domains are created for each instance of that class. For a domain `D` declared on a class `C` and two instances `o1` and `o2` of type `C`, the domains `o1.D` and `o2.D` are distinct, for distinct `o1` and `o2`.

Domain use. Each object is assigned to a single ownership domain that does not change at runtime. The extractor indicates the domain of an object by annotating each reference to that object in the program. For example, the `@Domain` annotation declares the reference `f` of type `InputFile` in the domain `L` (line 11).

The annotations define two kinds of object hierarchy, *logical containment* and *strict encapsulation*. *Logical containment* makes an object conceptually *part of* another, and is achieved using public domains. Having access to an object gives the ability to access all objects inside its public domains. For example, `IniFile` declares a public domain, `PARAGS`, to hold the field `para`. *Strict encapsulation* makes

```
1  @Domains({ "owned", "PARAGS" })
2  @DomainParams({ "U", "L", "D" })
3  public class IniFile {
4    @Domain("shared") String filename;
5
6    @Domain("owned<shared, PARAGS<U,L,D>>")
7      Hashtable<String, IniParagraph> paragraphs;
8
9    @Domain("PARAGS<U,L,D>") IniParagraph para;
10
11   @Domain("L") InputFile f;
12   ...
13 }
14 @Domains({ "owned" }) // != IniFile's owned
15 @DomainParams({ "U", "L", "D" })
16 public class IniParagraph {
17   @Domain("L") InputFile f;
18   ...
19 }
20 // Root class, used for OOG construction
21 @Domains({"UI","LOGIC","DATA"})
22 public class System {
23   // Outer LOGIC is the domain of the reference
24   // Class IniFile is parameterized with <U,L,D>
25   // We bind the domain parameters as follows:
26   // U := UI, L := LOGIC, D := DATA
27   @Domain("LOGIC<UI,LOGIC,DATA>")IniFile iniFile;
28   ...
29 }
```

Figure 1: Ownership domain annotations.

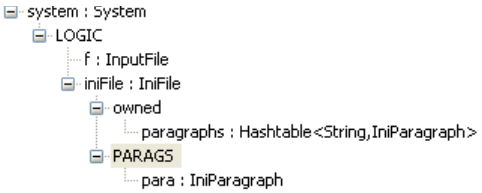
an object strictly *owned by* another, and is accomplished using private domains. For example, `IniFile` stores the `paragraphs` field in a private domain, `owned` (line 6). Strict encapsulation is stronger than the Java visibility mechanism. For instance, a `public` method cannot return an alias to an object inside a private domain, although the Java type system allows returning an alias to a field marked as `private`.

Domain parameters. Domain parameters on a type allow objects to share state. The extractor declares domain parameters using the `@DomainParams` annotation. For example, the class `IniParagraph` takes the `U`, `L` and `D` domain parameters, and these are bound to the corresponding domain parameters on `IniFile`, respectively. This way, both an `IniParagraph` object and an `IniFile` object can reference the same `InputFile` object in the `L` domain parameter (see incoming edges from `para` and `iniFile` to `f` in the OOG).

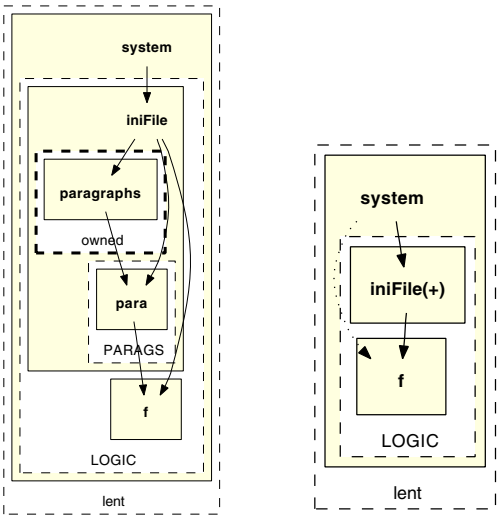
Special annotations. Special annotations add expressiveness to the type system [6]: `unique` indicates an object to which there is only one reference, such as a newly created object, or an object that is passed linearly from one domain to another. One ownership domain can temporarily lend an object to another and ensure that the second domain does not create persistent references to the object by marking it as `lent`. An object that is marked as `shared` may be aliased globally but may not alias non-`shared` references, and little reasoning can be done about it.

2.2 Object graph extraction

The extractor picks a root class as a starting point. The static analysis then uses the ownership annotations in the code to impose a conceptual hierarchy on the objects in the system. The root class in the example is `System` (Fig. 1). In the ownership tree (Fig. 2(a)), a low-level object such as `paragraphs` is underneath the more architecturally interesting object `iniFile`. Fig. 2(b) and Fig. 2(c) show the corresponding OOG of the above example (Fig. 1). Typically,



(a) Ownership tree, from the root object.



(b) Expanded OOG.

(c) Collapsed OOG.

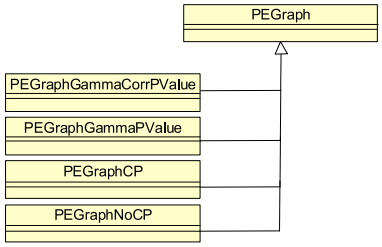
Figure 2: OOG for the above example (Fig. 1).

we do not show the root object and consider the domains declared on the root class to be top-level domains.

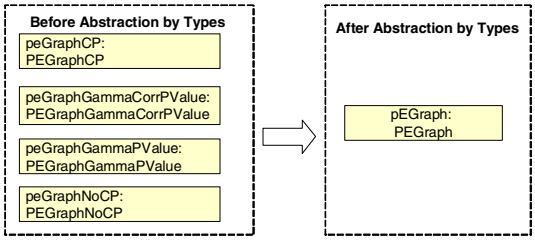
To construct the OOG, a static analysis abstractly interprets the program with annotations. In particular, it maps formal domain parameters to the actual domains to which they are bound. As a result, it shows an object declared inside the domain parameter `L` such as `InputFile` inside the actual domain `LOGIC` on the root object.

The visualization uses box nesting to indicate containment of objects inside domains and domains inside objects (Fig. 2). Dashed-border, white-filled boxes represent domains. Solid-filled boxes represent objects. Solid edges represent field references. An object labeled `obj:T` indicates an object reference `obj` of type `T`, which we then refer to either as "object `obj`" or as "`T` object", meaning "an instance of the `T` class". A private domain has a thick, dashed border; a public domain has a thin, dashed border. Having a hierarchical representation allows collapsing or expanding objects to control the level of visual detail. In Fig. 2(b), the OOG is fully expanded. In Fig. 2(c), we collapsed the sub-structure of `iniFile`. The symbol (+) on an object indicates that it has a collapsed sub-structure.

Abstraction by types. An OOG provides architectural abstraction primarily by ownership hierarchy. In addition, an OOG can abstract objects within each domain by their declared types. In object-oriented systems, many types extend from common base classes or implement common interfaces. To use abstraction by types, the extractor defines an ordered list of design intent types (DIT). To decide whether to merge two objects of type C_1 and C_2 in a domain, the analysis finds the first type, C, in the DIT such that C_1 is a subtype of C and C_2 is a subtype of C. If DIT does not include such a type, the objects are not merged (Fig. 3).



(a) Class inheritance hierarchy.



(b) Abstraction by types on OOG.

Figure 3: Result of applying abstraction by types.

3. SUBJECT SYSTEM

As our subject system, we chose Pathway Express, one of the Onto-Tools developed in the Intelligent System and Bioinformatics Laboratory at Wayne State University [15]. Pathway Express finds, builds, and displays a graphical representation of gene interactions. It has more than a thousand users spread across several research groups in bioinformatics. In the rest of this paper, we refer to Pathway Express as PX.

Development on PX has been ongoing since 2002 by a number of graduate students. The original developers graduated, and are no longer available, but the system is still actively maintained by other graduate students.

PX is an object-oriented web application implemented in J2EE which consists of 163 classes, 9 interfaces, and 30 packages, for a total of 36,000 lines of code, excluding libraries [13]. The maintainer set up a standalone project (with all the associated libraries) in the Eclipse development environment for the extractor to analyze.

Why this system? The SCHOLIA approach has a few characteristics that make it more challenging to evaluate. First, the approach requires access to the maintainers to capture their design intent. The PX system seemed like an acceptable candidate because we had access to its maintainers to help us refine and evaluate the OOG.

Second, the annotation process is currently mostly manual and carries a non-negligible overhead. As a result, we cannot complete the study by spending a day or two at a company. In our experience, getting access to professional programmers in commercial organizations or in open-source projects to evaluate research technologies, especially ones such as ours that are labor intensive, can be a challenge. One of our previous evaluations, a field study [2], required spending a full week on-site then following up to refine the OOG [5].

Third, the system already has some documentation: a few entity-relationship diagrams that explained the database schema and a high-level diagram [7, Figure 5.2] that shows

that the system follows the client-server architectural style. The high-level diagram does not show the detailed structure of the components in the user interface or the logic tiers. In contrast, the OOG will represent the runtime architecture in more detail. Although the OOG has those details, it supports both high-level and detailed understanding, due to the ability to collapse or expand the sub-structures of selected objects.

Finally, several maintainers of the PX system reported experiencing much difficulty in understanding and evolving the system. So, the maintainers may benefit from the additional documentation that the OOG provides, since as a diagram of the runtime architecture, the OOG complements the available documentation of the code structure and augments the high-level architectural diagram.

4. EXTRACTION

The extraction phase of the study included adding the initial annotations, fixing the annotations to address the typechecker's warnings, extracting OOGs and refining them.

4.1 Adding the initial annotations

The extractor needed to determine the top-level domains of the system, and how to map the objects in the system to the various domains.

Determining the top-level domains. From an initial meeting with the system maintainer, the extractor decided to organize the application into three tiers, User Interface, Logic, and Data, which he represented using top-level domains, `UI`, `LOGIC`, and `DATA`, respectively.

Mapping objects to domains. Ideally, the system maintainers provide the mapping of objects to domains or refer the extractors to any available documentation that describes the system structure, textually or using diagrams. But since the extractor did not have such information for PX, he relied on the names of packages or types, to determine to which domain the instances of the type belong. For example, he used the package name `oe.standalone.client.pe.gui.oe` as a hint that the instances of the types declared in that package, such as `DefaultFunctionBar`, are in the `UI` domain. Similarly, based on the type name, he placed instances of the type `DBConnectionManager` in the `LOGIC` domain, and instances of `OntoToolData` in the `DATA` domain.

Defining domain parameters. Once the extractor determined the three top-level domains, `UI`, `LOGIC`, and `DATA`, he defined the corresponding domain parameters `U`, `L` and `D`, respectively. These domain parameters will be added to the various types, and allow object sharing across domains.

Mapping types to domains. In order to propagate the initial set of annotations, we developed a defaulting tool, hereafter called ArchDefault, to add more of the annotations automatically. As a first approximation of mapping objects to domains, the input to ArchDefault is a map from types to domains. The map assumes that all the instances of one type are in the same domain, which is not always the case. For example, different instances of `ArrayList` are typically inside different objects. As a result, we typically leave out of the map the types of data structures, such as `ArrayList`.

Based on this map, ArchDefault generates some initial annotations in the code. Then the extractors complete the annotations by fixing annotation warnings and refining the annotations. From the map alone, ArchDefault tends to generate a graph where many objects are in the top-level domains. For example, it maps all instances of the type `IniParagraph` to `L`, which means that the `para` object will appear in the `LOGIC` top-level domain. To reduce the clutter in the `LOGIC` domain, the extractor can push the `para` object inside a public domain declared in an `iniFile` object. To do so, the extractor manually declares the `PARAGS` public domain on the `IniFile` class, and changes the annotation on the `para` field inside the class `IniFile` from `L` to `PARAGS`. Indirectly, the extractor is mapping objects to domains.

4.2 Extracting and refining OOGs

Adding annotations. The extractor relied mostly on the typechecker to guide the annotation process. The typechecker validates that the annotations are consistent with each other and with the code, and raises annotation warnings. The extractor addressed the high-priority annotation warnings, and ended up with 469 warnings. The remaining warnings are due to inherent expressiveness challenges in the type system, bugs in the tools that we are currently fixing, or design issues in PX.

Determining the root class. The extractor had to determine the root class, from which the OOG extraction tool starts analyzing the annotated code. There were around 20 `static void main` methods scattered in different classes of the system. Most of these methods were used as test harnesses for unit testing. We found two likely root classes, `PathwayExpress` and `OntoToolsApp`.

We first chose `OntoToolsApp` as the root class. However, we realized that the extracted OOG was missing one of the top-level domains and many objects in the top-level domain.

The extractor is not a J2EE web application expert but knows that each client-server application has two main components. The server-side component runs on the host server. In J2EE, this component is usually an instance of the type `Servlet` or any sub-type thereof, such as `HttpServlet`, that responds to the client's requests. The extractor found the class `PathwayExpressServlet` which subclasses `HttpServlet`, as a server-side type.

The client-side component is typically a Java applet which requests services from the server and displays the results in the client's web browser. For PX, the extractor found the class `PEInputApplet`, which represents the client-side and instantiates the root class `OntoToolsApp`, which in turn initializes several of the core objects in the system.

PX is a web-based system, so the instantiation of objects on the server is controlled by various configuration files, which are interpreted by the server infrastructure. But our static analysis does not know about these files. So, the extractor had to simulate the initialization of the system and provide this information explicitly to the static analysis.

The goal was to extract an architecture of the entire, complete system, so we wanted to include both client- and server-side objects. So, the extractor created a top-level class, `Main`, which instantiates `PathwayExpressServlet` on the server side, and `PEInputApplet` on the client side, to include all the system objects.

Extracting and refining initial OOGs. The extractor used the OOG extraction tool and extracted some initial OOGs. The initial OOG displayed many objects in the top-level domains, and some of these objects were low-level, implementation details. The extractor can control the abstraction by ownership hierarchy in the OOG using the annota-

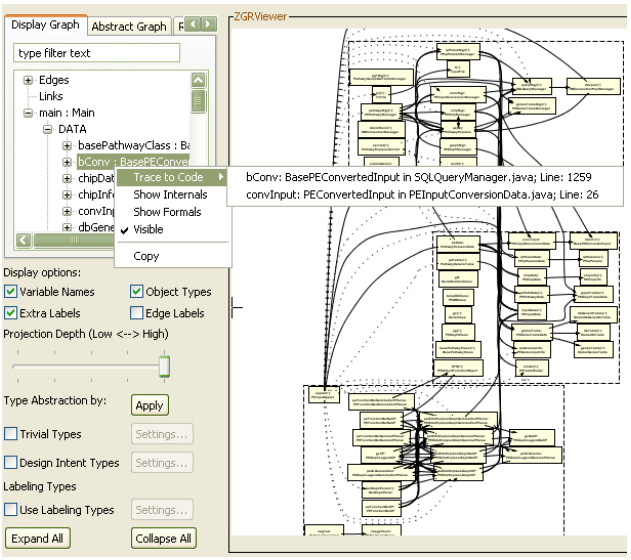


Figure 4: OOG tool used by the maintainer.

tions. Namely, he can push a low-level object underneath a more architecturally significant objects. To do so, he can:

- **Make an object *owned by* another.** For example, he moved the objects of type `PEGraphClass` from the top-level domain `D` to the domain `owned` inside the class `PEGraphTableData`; or
- **Make an object *part of* another.** For example, he created the public domain `PARAGS` inside the class `IniFile`, then moved the object `para:IniParagraph` from the top-level domain `DATA` into the `PARAGS` domain (Fig. 1). As another example, the extractor declared the public domain `DB` in the type `PETableModel` to hold objects of type `Row`. Since several public methods on `PETableModel` returned objects of type `Row`, the latter could not be placed into a `private` domain.

5. EVALUATION

In the second phase of the study, we evaluated the extracted OOG with the maintainer of the system. The goal of the evaluation was to study whether the maintainer understood the extracted OOG, and whether he can suggest his own refinements using the two types of abstraction, i.e., abstraction by ownership hierarchy and by types, to make the OOG reflect his mental model of the system.

5.1 Method

Participant. We asked the lead maintainer of the system, a PhD student in the department of computer science at Wayne State University, to evaluate the extracted OOG. The maintainer was already familiar with the Eclipse integrated development environment.

Tools and Instrumentation. We gave the participant the OOG extraction and viewing tool, which is an Eclipse plugin that allows navigating the OOG interactively (Fig. 4). We used Camtasia to record the think-aloud of the participant and to capture the screen of the participant while he used the OOG tool and the Eclipse environment.

Procedure. The extractor launched the tool and displayed the extracted OOG. Then, he gave the maintainer a quick tutorial explaining the tool and its features. The extractor asked the maintainer a set of questions (Table 1). To answer each question, the maintainer went through each tier, one tier at a time, and each object in that tier. The meeting with the maintainer suggested further refinement of the OOG. Therefore, the extractor incrementally refined the extracted OOG until it matched the maintainer's mental model.

Table 1: Questions asked to the maintainer.

No.	Question
Q1	Does object X of type T belong to tier A? And if not, to which tier does it belong?
Q2	Which objects, do you think, are useful and helpful for code modifications to see at the top-level of the OOG?
Q3	Are there any missing objects from the top-level of the OOG? or from the rest of the OOG?

Table 2: Main OOG refinements.

No.	OOG Refinement
R1	Move an object between sibling domains
R2	Abstract a low-level object
R2.1	Make an object conceptually *part of* another object
R2.2	Make an object *owned by* another object
R3	Move an object to higher-level domain
R4	Collapse related instances of subtypes

5.2 Analysis

We transcribed the think-aloud recordings and screen capture video into action logs, consisting of a total of 319 lines. In a previous case study [5], we had developed a classification of the possible refinements that can be performed on an extracted OOG (Table 2). In this study, we used qualitative protocol analysis, and coded the refinements that the maintainer requested using the same classification model.

5.3 Results

We now present the results of the evaluation of the OOG based on both analyzing the maintainer's responses to our questions (Table 1) and the ability of the extractor to further refine the OOG to reflect the maintainer's mental model. We also report the extractor's effort in the extraction and refinement of the OOG.

One feature of the statically extracted OOG is that is has traceability from the diagram to the code. During the evaluation of the OOG by the maintainer, and during the subsequent refinement of the OOG, both the maintainer and the extractor traced from a selected node or edge in the OOG to the corresponding lines code to understand why the OOG displayed a certain object or edge, or to change the annotations in the code to refine the OOG.

Maintainer's answers to the questions. At the beginning of the meeting, the maintainer could not give accurate answers to the questions, since he did not know about all the objects in the system. To help him answer the questions better, he used several features of the OOG extraction tool. For example, he traced from objects or edges on the OOG to the corresponding lines of code. He also found the ownership tree-view useful (left-hand side of Fig. 4) to navigate all the objects on the OOG.

For question Q1, the maintainer navigated all objects in the ownership tree and mostly agreed with how the extractor had assigned the objects to domains. In a few cases, the maintainer suggested moving some objects from one domain to another. For example, he suggested moving the object `model:PETableModel` from `UI` to `DATA`. In other cases, he was

Table 3: Refinements requested by the maintainer.

Refinement	Requested	Completed
R1	19	18
R2	40	21
R3	0	0
R4	20	14

Table 4: Number of objects in the top-level domains.

Domain	At meeting	After meeting
`UI`	29	7
`LOGIC`	12	16
`DATA`	27	23
Total	68	46

unsure in which domain an object should be. For example, `f:InputFile` was in `DATA`, and he thought that it could also be in `LOGIC`. For some cases, he could not confirm the domain of an object just by looking at the OOG, so he traced to the code to better understand how the object was being used.

For question Q2, the maintainer navigated all the top-level objects in the OOG carefully. In a few cases, he pointed out some objects that seemed too low-level to be in the top-level domains. He also suggested merging all the objects that have `Table` in their declared types, like `pdTable: PathwayDetailsTable`, `ipIdGenesTable: InputIdGenesTable` and `model: PETableModel`.

For question Q3, the maintainer wondered why some objects that display the computation results were missing from the OOG. While the maintainer navigated the OOG, he asked questions about some edges between some objects, such as the edge between `lf:LoginFrame` and `otApp:OntoToolsApp`. The extractor traced from this edge to the code. The corresponding field declaration in the code helped him understand why this edge was in the OOG. The maintainer also reported that some edges were missing from the OOG, such as the edge between `lf:LoginFrame` and `inputFrame:PEInputFrame`. He traced from the `inputFrame:PEInputFrame` object on the OOG to the code. The code did not have the field declaration that the maintainer expected to see, which explained why the edge was missing from the OOG.

OOG refinements requested by the maintainer. Interestingly, the maintainer requested that many objects (19 objects) be moved between top-level domains (**R1**, Table 3). This is unsurprising since we did not have the maintainer validate the map before we proceeded to add the annotations. Also, he did not recommend any objects be moved up to the top-level domains (**R3**), presumably because the top-level domains were still too cluttered.

We measured the number of objects in the top-level domains, before meeting with the maintainer and after we refined the OOG (Table 4). We were able to reduce the number of objects in the top-level domains (from 68 to 46). Still, we fell short of the rule of thumb in architectural documentation to have 5 to 7 components per tier [16]. In particular, we still have a high number of objects in the `DATA` domain, due to our inability to apply additional abstraction by types. For instance, we can recommend to the maintainers that they create a common supertype for a number of related types that contain `Table` in their name, because the maintainer thought these objects were related (Section 5.3).

Refinement of the OOG to implement the maintainer's requests. Prior to meeting with the maintainer, the extractor did not use abstraction by types, because he could not easily determine which types were architecturally significant in order to add them to the list of design intent types. Moreover, in PX, there was not a single Java package that held all the core interfaces or abstract base classes. Instead, they were scattered across several Java packages.

Table 5: Time to extract the OOG.

Phase	Hours	Percent
Adding annotations and extracting OOGs	31	69%
Building the ArchDefault map	5	11%
Refining the OOG on our own	5	11%
Meeting with the maintainer	1	2%
Refining the OOG after the meeting	3	7%
Total	45	100%

To implement the maintainer's requests, the extractor used both abstraction by ownership hierarchy and abstraction by types (Table 3). To apply abstraction by ownership hierarchy, he modified the annotations in the code to push low-level objects underneath others (**R2**). To apply abstraction by types (**R4**), the extractor selected a number of architecturally interesting types. For example, he specified the `FunctionBar` type as a design intent type, so the OOG grouped several objects of types `FunctionBar`, `PEFunctionBar`, and `PEFunctionBarGammaPValue` in the top-level domain `UI`, because all of their types share the common super-type `FunctionBar`. In some cases, the extractor could not implement the maintainer's requests since the code did not support the change. For example, the maintainer suggested merging all objects that have `Table` in their declared types, such as `pdTable: PathwayDetailsTable`, `ipIdGenesTable: InputIdGenesTable` and `model: PETableModel`. Much to the maintainer's surprise, the types of these objects did not have a common super-type. As a result, the extractor could not use abstraction by types to accomplish this merging.

Measured effort of the extractor. We measured the extractor's effort spent during both the extraction and refinement phases. The extraction phase consisted of adding the annotations and fixing the annotation warnings. Refining the extracted OOG involved changing the annotations, incrementally and in a localized manner. When the extracted OOG did not match the maintainer's mental model, the extractor had to identify the cause of the discrepancy, and change the annotations in the code consistently then re-extract the OOG.

The extractor spent 31 hours adding annotations and fixing warnings (Table 5), and that puts the extraction effort at roughly 1-hour/KLOC. The extractor spent around 5 hours on his own refining the OOG, before showing it to the maintainer. After meeting with the maintainer, he spent another 3 hours refining the OOG. The meeting with the maintainer lasted around one hour, which is 2% of the total time. So the approach did not require significant and continuous involvement from the maintainers' part.

6. DISCUSSION

We revisit the research questions and discuss how well this study answered them. We then discuss the effectiveness of the ArchDefault tool, some lessons learned from adding the annotations and from extracting OOGs, some limits to the abstraction by ownership hierarchy and by types, some threats to validity. and some limitations.

Table 6: Frequency of the annotations.

Annotation	Frequency	Percent
`U`	125	2.2%
`L`	75	1.3%
`D`	511	9.1%
`owned`	278	4.9%
`shared`	2,994	53.1%
`unique`	363	6.4%
`lent`	1,273	22.6%
Public domains	6	0.1%
Top-level domains	3	0.1%
Other domain parameters	6	0.1%
Total	5,634	100%

6.1 Research Questions

Regarding RQ1, the extractor was able to use abstraction by ownership hierarchy and by types to obtain an OOG that was at an acceptable level of abstraction, prior to meeting with the maintainer.

Precision of the OOG. We believe that the extracted OOG is reasonably precise. We attribute this to the precision of the annotations and the number and severity of remaining annotation warnings. Also, having imprecise types in the code can lead to imprecise edges in the OOG, i.e., more edges than are needed for soundness. We mitigated this by refactoring the code to generics before adding annotations. There were a few cases where the extractor could add only generic types that are imprecise such as `Object`, `Serializable` or `Cloneable`. In fact, this indicated some design smells in the PX code base. Indeed, code that cannot be easily genericized often cannot be easily annotated with Ownership Domains. This should come as no surprise since some ownership type systems combine ownership types and generic types [11].

Annotation Metrics. We computed some simple metrics on the annotations we added (Table 6) to judge the quality of the annotations. For example, a high proportion of `shared` annotations meant that we were not reasoning about many objects in the system. In addition, we typically do not show the `shared` domain in an OOG and the objects inside it. In PX, there are 5,634 fields or variables of a reference type which have annotations. Of those, 53% are `shared`, which is high and is due to the excessive use of `String` objects.

Regarding RQ2, we believe that the maintainer understood the notion of hiding low-level objects underneath more architecturally-relevant ones. He requested 40 cases of moving an object from a top-level domain underneath some other object. In most cases, he did not specify to which domain the extractor should move the object. He also specifically requested combining several objects, thinking that they were related. Some of these objects were not always related by subtyping. Instead, the maintainer thought that their classes were related by a loose naming convention. There were 20 such cases that we grouped into 3 clusters of related objects with 7, 8 and 5 objects, respectively.

During the meeting, the maintainer demonstrated that he both understood the OOG and used the OOG tool effectively. For example, when the extractor asked the maintainer: *"Can you see all objects you expect to see in the extracted OOG?"*, the maintainer said: *"No, there is a missing object that displays PX results."* The maintainer then started searching for that object on the OOG. He located an object that he thought should be connected to the missing object, and traced to the code. He similarly navigated the OOG to investigate some edges that he did not expect to see.

Clash with the maintainer's expectations. We did face a challenge in encouraging the maintainer to think in terms of the runtime structure rather than the code structure. For instance, he wanted to split the `PEGUIManager` object across two tiers in the OOG, `LOGIC` and `UI`. In Scholia, an object is in exactly one domain. Furthermore, one runtime object has exactly one representative in an OOG. It would be misleading to have one runtime object appear as two boxes (two components) on an architectural diagram. Then one could assign to each component a different value for a key architectural property and potentially invalidate the analysis results. So the extractor was unable to accommodate this request, without refactoring the code.

Regarding RQ3, the extractor was able to refine the OOG to reflect the maintainer's design intent. Indeed, he was able to address most of the maintainer's changes (see the *Completed* column in Table 2). He was able to perform the refinements without changing all the annotations already in place. Moreover, the refinement phase took significantly less time than the extraction phase (3 vs. 31 hours in Table 5).

Clash with the maintainer's expectations. As discussed above with the objects that have `Table` in their declared types, the extractor was unable to apply the abstraction by types in a few cases where the maintainer thought that some objects were related. Upon further investigation, however, the extractor found that the classes of these objects do not share a common super-class.

6.2 Effectiveness of the ArchDefault tool

Using ArchDefault in the beginning helped the extractor start with fewer warnings to manually resolve, then reach an acceptable level of warnings more quickly, in order to extract initial OOGs then refine the OOG with the maintainer. Thus, it was possible to use a tool to propagate many of the annotations automatically. The area where ArchDefault helped the most was in propagating domain parameters, including dealing with domain parameters and inheritance. ArchDefault is not a smart ownership inference tool, however, and the extractor still had to manually fix many annotation warnings. Reducing the number of warnings in the annotated system is crucial. Scholia guarantees the OOG's soundness only if the program is fully annotated and the annotations typecheck without warnings.

During the study, we thought of a few improvements to ArchDefault. For instance, it would be nice to map all instances of all the types defined in a package to one domain. For example, the types `Font`, `Image` and `Point` are in the `java.awt` package. Instead of listing many fully qualified type names in the map, it may be preferable to use regular expressions to map all the types in the package `java.awt` to the `shared` domain.

6.3 Lessons learned from adding annotations

We learned that it is hard to add annotations to code that lacks generic types or to code that is hard to refactor to generics. This should come as no surprise since some ownership type systems combine ownership and generic types [11]. To avoid breaking the code in some cases, the extractor made a generic container take an imprecise type argument such as `Object`, `Serializable` or `Cloneable`. These imprecise declared types in the code can also lead to imprecise edges in

```
@Domains({ "owned" })
@DomainParams({ "U", "L", "D" })
abstract class AbstractStandaloneInput ... {
  protected @Domain("owned<D>")
  List<Serializable> references;
  ...
  public @Domain("unique<D>")
    List<Serializable> getReferences() {
     //TODO: Return a copy of the owned field.
     return references;
  }
  ...
}
```

Figure 5: Warning about representation exposure.

the OOG, i.e., more edges than soundness requires.

The annotations revealed many instances of representation exposure, which gives external code access to the private state of an object and breaks encapsulation. In legacy code, this is very common: developers define a `private` field to hold internal state, then define a `public` method that returns an alias to the object stored in the `private` field. Such code is legal based on Java's visibility modifiers. One solution is to return a shallow copy of an internal list instead of an alias. We added TODOs in the code as reminders for the maintainers to investigate and fix such cases (Fig. 5).

6.4 Lessons learned from extracting OOGs

Lack of rich inheritance hierarchy. As discussed above, we could not apply the abstraction by types in a few cases. Several of the classes that the maintainer thought to be related did not share a common superclass. This unexpected inheritance could be a design smell or it could indicate possible code duplication. Ideally, the common code will be refactored and pushed up into the super-class.

Loosely-typed containers. We also found several loosely-typed containers which store objects which do not share a common type that is more specific than `Object` or `Serializable`.

Lack of user-defined types. Similarly, the code heavily uses `String`s instead of user-defined types. In our approach, we treat `String`s as `shared`, do not reason about them, or even show the `shared` domain on the OOG. From a design standpoint, it is better to declare and use a user-defined type, e.g., a type `GeneUniqueId` instead of `String`.

Overall, these design issues point to the lack of a rich type structure and the dearth of precise, declared types in the code. Not only can the OOG enable the maintainers to identify these design smells, it can also guide the maintainers toward fixing the issues. For instance, the maintainers can create a common supertype for a number of related types that contain `Table` in their name.

6.5 Limits of abstraction

Despite our efforts, the PX OOG is still too cluttered. For example, we wanted to use abstraction by types more heavily, but were limited by the system design. Refining the PX OOG further would have required re-engineering the code, sometimes in simple ways. For example, we could have defined empty interfaces, made related classes implement those interfaces (i.e., *use marker interfaces to define types* [8, Item#37]), then added the interfaces to the list of design intent types to refine the abstraction by types. Other re-engineering would be more complex.

Other possible abstraction mechanisms. The challenges in achieving an uncluttered graph with PX suggest that additional abstraction mechanisms, beyond abstraction by ownership hierarchy and types, might be necessary, especially as OOGs are applied to larger and larger systems. One idea, for example, is to allow "abstraction by name", to merge, without changing the inheritance hierarchy, several objects that have `Table` in their declared types, like `pdTable: PathwayDetailsTable`, `ipIdGenesTable: InputIdGenesTable` and `model: PETableModel`.

More generally, the OOG can allow "abstraction by purpose". After applying abstraction by ownership and types, extractors might specify groupings among the remaining objects by their conceptual purpose, even if they are unrelated by type or by name. This is similar to the manual specification of task-specific abstractions in approaches such as Reflexion Models [19].

6.6 Threats to Validity

Internal Validity. One may argue that the results could be largely due to the extractor's knowledge of the PX code, rather than to following the Scholia approach. This was not the case for the PX study. In fact, during the meeting, the lead maintainer admitted that he was not thoroughly familiar with all the objects in the system. In Scholia, the extractor is guided mainly by the typechecker. If he adds an annotation that the code does not support, the typechecker issues a warning that serves as a reminder.

External Validity. Admittedly, PX is not representative of all object-oriented systems, so it is hard to generalize the results from this case study. PX is not the only medium-sized system we analyzed, however. Abi-Antoun and Aldrich [2] previously conducted a field study to extract the architecture of a 30-KLOC module, called LbGrid. There are several important differences between the two studies.

For the PX study, we developed and used ArchDefault to generate the initial annotations. For LbGrid, we used the previous defaulting tool. At the end of the first phase with LbGrid, we had 4,000 warnings, whereas PX has fewer than 500 warnings. We attribute this improvement largely to the effectiveness of the ArchDefault tool at reducing the manual annotation burden. For PX, we incorporated the maintainer's refinements by changing the annotations to control the abstraction by ownership hierarchy. We also used abstraction by types to declutter the OOG. Unlike the LbGrid case study, the PX maintainer interacted directly with the extraction tool. From an external validity standpoint, LbGrid was implemented mostly by one professional programmer and already used generics, whereas the PX code was implemented by several graduate students.

In the PX study, unlike the LbGrid study, the extractor was not one of the Scholia designers. The PX extractor was a first-year Ph.D. student who has an M.S. degree, which makes him somewhat representative of an entry-level, professional programmer. PX was the first large system that he analyzed, after receiving a tutorial, practicing on several small examples (around 200 LOC each), adding annotations and extracting OOGs from a much smaller system (1.4 KLOC). With proper training, outside extractors can use the approach and the tools on other subject systems.

Also, for PX, we tracked the extractor's effort much more precisely than for LbGrid. For LbGrid, we only measured the aggregate end-to-end effort (35 hours). For PX, we mea-

sured the effort for adding initial annotations and extracting initial OOGs, separately from the effort of refining the OOG based on the maintainer's feedback. This way, we were able to measure that the refinement effort is much less than the effort to extract an initial OOG.

6.7 Limitations

Limitations in the study design. The extractor did not have any documentation to help him map objects to domains, and he did not have the maintainer verify the mapping before proceeding. As a result, during the OOG refinement, the extractor had to re-map many objects to different domains, which increased the overall extraction time.

For future studies, it may be preferable to have the maintainers provide or validate the initial mapping, instead of having the extractor propose a mapping, use it to add annotations, then have the maintainers validate the placement of objects into domains. However, the maintainers might be too intimidated by a novel approach, and not know what to do. It was also a good sign that the extractor was able to tweak the annotations after the fact, with relative ease and without having to wipe them all out and redo them.

Scholia does not currently support the interactive refinement of the OOG by direct manipulation. One idea for a future study would be to give the maintainers an interactive editor and have them make the changes directly to the OOG. We already have a prototype interactive editor [4], but it still suffers from usability issues. In addition, the key challenge is maintaining the soundness of the OOG.

Limitations in Scholia. There are still annotation warnings in PX. Some warnings are due to inherent expressiveness challenges in the ownership type system. For example, static code is challenging to most ownership type systems, which track instances. Other warnings are due to bad code in PX. For instance, there are several cases of representation exposure, where a public method returns an alias to a private field. These are precisely the mistakes that ownership types are designed to prevent.

The scale of the system we analyzed may pale in comparison to previous case studies that analyzed the code architecture of large systems. Indeed, the tools that analyze the runtime architecture are much less mature than those that analyze the code architecture. Moreover, Scholia is a design-intent-based technique that requires specifying the design intent using annotations. This makes using Scholia to analyze millions of lines of code prohibitively costly, without automated annotation inference. For comparison, the closest prior work that used annotations to extract object models by Lam and Rinard [17] was evaluated on one 1,700-line system.

7. RELATED WORK

Similarities with Reflexion Models. Murphy's Reflexion Models (RM) [19] inspired the idea of using a map for ArchDefault. In RM, a developer maps source-level entities to components in a high-level model. For instance, he maps an interface "Foo" to a "manager" component. In our approach, the extractor indirectly maps objects to domains. Using RM and building correctly by hand an entire map of objects to higher-level components is challenging since the map has to deal with aliasing, inheritance and polymorphism [1, See §6.6.4 for an interpretation of RM].

General architectural extraction process. There are several published case studies in architectural extraction, some of which tackled big legacy systems written in procedural languages. Tzerpos and Holt [25] used a "hybrid" process that combines facts extracted from the code and information derived from interviewing developers. These steps include: collecting "back of the envelope" designs from project personnel; extracting raw facts from the source code; collecting naming conventions for files; clustering code artifacts based on naming conventions; creating tentative structural diagrams, and collecting the reactions of the developers to these tentative diagrams; and so on, until they converged to a code architecture. They concluded that there is a reasonably well-defined sequence of steps to go through to extract a code architecture. Indeed, we followed similar steps to extract the runtime architecture, but did not use clustering and relied on the maintainer to a much smaller extent.

Many architectural extraction studies use various sources of information that are extrinsic to the code, with no clear success criteria. It is also fairly common for different architectural extraction teams to produce very different architectures. In Scholia, the annotations have to be consistent with each other and with the code, and are checked using a tool. During the PX study, we followed a principled approach: we added annotations, typechecked them, extracted OOGs, refined the OOGs on our own, then involved the system maintainer to a very limited extent (Table 5). Our success criteria are to minimize the number of objects in the top-level domains, and the number of annotation warnings.

Architecture extraction of procedural systems. Gröne et al. [14] manually extracted the architecture of Apache (written in C) and represented the architecture using agents. The architectural extraction involved ad-hoc manual techniques and many people—many students enrolled in a class. The only tool used for the analysis of the source code transformed the C source code into a set of syntax highlighted and hyperlinked HTML file. The study justified not using more advanced tools by saying that "an important amount of information needed for the conceptual architecture is not existent in the code and therefore cannot be extracted by a tool" [14]. This is precisely why Scholia uses annotations, to clarify some of the architectural intent in the code, and extract abstractions that match the mental model of the system maintainer.

Extraction of code architecture of object-oriented systems. Several researchers extracted architectural views of large object-oriented systems, e.g., Jigsaw system (300 classes), but focused on the code architecture [18, 9]. Medvidovic and Jacobak [18] point out that tools are often unable to discover a relationship that is implemented indirectly, e.g., by using instances of container classes such as `Vector`, `Map`, `List`, to store objects of some other application class. User-defined container classes complicate matters further. On the other hand, Scholia readily handles those container classes. Indeed, we spent a lot of effort adding generic types first, then ownership domain annotations to the containers. In particular, when adding annotations to a container, one has to specify separately the annotation on the container object itself and the annotation on the contained elements.

Dynamic extraction of runtime architecture. Many have preferred dynamic analyses (or mixed with static analysis) to extract the as-built runtime architecture [20, 21].

Our approach is entirely static, so we did not have to run the PX system. Running this system will have significantly complicated the extraction process, since it will have required setting up a database server, a test database, a web server, and the right configuration files, among others.

Evaluation of extracted architectures. Evaluating the quality of an extracted architecture is subject to debate, with no generally accepted evaluation criteria. More generally, this appears to be a common issue in the empirical evaluation of reverse engineering tools. Tonella et al. [24] state: "the same piece of information recovered from the code may be immensely useful or completely unusable depending on the end user who is performing the current software engineering task and depending on the amount of knowledge the user already has about the system".

One approach to measure the "goodness" of an extracted architecture is to compute various structural metrics. Indeed, clustering methods use this approach to evaluate the quality of the result. For example, a clustering is "good" if the clusters are reasonably sized and exhibit low coupling and high cohesion [22]. In our case, we measured percentages on the annotations we added, and ensured that we were not treating an excessive proportion of the objects as `shared` (Table 6). Another measurable success criterion was to minimize the objects in the top-level domains (Table 4).

8. CONCLUSION

In this paper, we presented the results of a case study in extracting the runtime architecture of an actively maintained object-oriented system. We confirmed that the system maintainers understood a global, hierarchical object graph of the entire system, and that it was possible to refine the OOG incrementally to convey their design intent.

There were a few cases where the as-built OOG clashed with the maintainer's expectations. Addressing these clashes will likely require refactoring the code to fix some design smells. For example, we showed how implementation or design decisions influenced the quality of the extracted architecture: the poor inheritance hierarchy prevented us from using abstraction by types, as much as we would have liked. In future work, we will observe how the maintainers use the extracted OOGs while evolving the system, to evaluate the benefit of the additional architectural documentation.

Acknowledgements. The authors thank Professor Sorin Draghici for granting us permission to study the Pathway-Express code base, and for making his graduate students available to evaluate the extracted OOG.

9. REFERENCES

[1] M. Abi-Antoun. *Static Extraction and Conformance Analysis of Hierarchical Runtime Architectural Structure*. PhD thesis, Carnegie Mellon University, 2010. Technical Report CMU-ISR-10-114.

[2] M. Abi-Antoun and J. Aldrich. A Field Study in Static Extraction of Runtime Architectures. In *PASTE*, 2008.

[3] M. Abi-Antoun and J. Aldrich. Static Extraction and Conformance Analysis of Hierarchical Runtime Architectural Structure using Annotations. In *OOPSLA*, 2009.

[4] M. Abi-Antoun and T. Selitsky. Interactive Refinement of Runtime Structure. In *Workshop on Flexible Modeling Tools (FlexiTools)*, 2010.

[5] M. Abi-Antoun, T. Selitsky, and T. LaToza. Developer Refinement of Runtime Architectural Structure. In *Workshop on SHAring and Reusing architectural Knowledge (SHARK)*, 2010.

[6] J. Aldrich and C. Chambers. Ownership Domains: Separating Aliasing Policy from Mechanism. In *ECOOP*, 2004.

[7] K. Amin. Pathway-Express: a Bioinformatics Tool for Pathway Level Analysis using Gene Expression Data. Master's thesis, Wayne State University, 2007.

[8] J. Bloch. *Effective Java*. Addison-Wesley, 2001.

[9] S. Chardigny, A. Seriai, M. Oussalah, and D. Tamzalit. Extraction of Component-Based Architecture from Object-Oriented Systems. In *WICSA*, 2008.

[10] P. Clements, F. Bachman, L. Bass, D. Garlan, J. Ivers, R. Little, R. Nord, and J. Stafford. *Documenting Software Architectures: View and Beyond*. Addison-Wesley, 2003.

[11] W. Dietl, S. Drossopoulou, and P. Müller. Generic Universe Types. In *ECOOP*, 2007.

[12] S. Ducasse and D. Pollet. Software Architecture Reconstruction: A Process-Oriented Taxonomy. *TSE*, 35(4), 2009.

[13] Eclipse Metrics Plugin. `http://metrics.sourceforge.net/`, 2010.

[14] B. Gröne, A. Knöpfel, and R. Kugel. Architecture Recovery of Apache 1.3 – a Case Study. In *International Conference on Software Engineering Research and Practice*, 2002.

[15] Intelligent Systems and Bioinformatics Laboratory. `http://vortex.cs.wayne.edu/projects.htm/`, 2003.

[16] H. Koning, C. Dormann, and H. van Vliet. Practical Guidelines for the Readability of IT-Architecture Diagrams. In *SIGDOC*, 2002.

[17] P. Lam and M. Rinard. A Type System and Analysis for the Automatic Extraction and Enforcement of Design Information. In *ECOOP*, 2003.

[18] N. Medvidovic and V. Jakobac. Using Software Evolution to Focus Architectural Recovery. *ASE*, 13(2), 2006.

[19] G. Murphy, D. Notkin, and K. J. Sullivan. Software Reflexion Models: Bridging the Gap between Design and Implementation. *TSE*, 27(4), 2001.

[20] T. Richner and S. Ducasse. Recovering High-Level Views of Object-Oriented Applications from Static and Dynamic Information. In *ICSM*, 1999.

[21] B. Schmerl, J. Aldrich, D. Garlan, R. Kazman, and H. Yan. Discovering Architectures from Running Systems. *TSE*, 32(7), 2006.

[22] T. Systä, P. Yu, and H. Müller. Analyzing Java software by combining metrics and program visualization. In *CSMR*, 2000.

[23] P. Tonella and A. Potrich. *Reverse Engineering of Object Oriented Code*. Springer-Verlag, 2004.

[24] P. Tonella, M. Torchiano, B. Du Bois, and T. Systä. Empirical Studies in Reverse Engineering: State of the Art and Future Trends. *Empirical Software Engineering*, 12(5), 2007.

[25] V. Tzerpos and R. C. Holt. A Hybrid Process for Recovering Software Architecture. In *CASCON*, 1996.